Test-Driven Development of Embedded Software

Boydens, Jeroen[ab][1,2] , Cordemans, Piet[c][1], Steegmans, Eric[d][2]

January 27, 2010

[1]"KHBO" Dept of Industrial Engineering Science & Technology
Zeedijk 101, B–8400 Oostende, Belgium

[2]"K.U.Leuven" Dept of Computer Science
Celestijnenlaan 200A, B–3001 Leuven, Belgium

{Jeroen.Boydens, Piet.Cordemans}@khbo.be
{Jeroen.Boydens, Eric.Steegmans}@cs.kuleuven.be

Abstract

Software in embedded systems plays an essential role. Principles of Test-Driven Development can be applied to increase quality of source code. TDD promotes the creation of code driven by automated tests. TDD exists for many years in general software development. This paper focuses on the migration of TDD to embedded system development. However, embedded systems do not lend themselves towards test automation. There are a number of obstacles when applying TDD on embedded systems. Amongst others is the strong hardware dependency of software, or even hardware that is missing at software development launch time. The paper shows that programming to an interface, instead of to a concrete class itself, isolates the software from the hardware. Then virtual drivers, which mock the behavior of real hardware drivers, can be implemented to test business logic behavior. Another option is to include a test wrapper on the embedded system. Based on existing measurements of advantages of TDD in general software development, the paper experiences comparable when applying the TDD methodology to embedded system design. Not only decreases the number of software bugs, furthermore the project's life cycle shortens.

Keywords: *embedded software, test-driven development*

---

[a] J. Boydens is a professor in Software Engineering at KHBO

[b] J. Boydens is an affiliated researcher at K.U.Leuven, dept CS, research group SOM

[c] P. Cordemans is staff member at KHBO funded by IWT-090191

[d] professor E. Steegmans is head of research group SOM, dept CS, K.U.Leuven

1 Introduction

As embedded systems are currently becoming more and more complex, the importance of their software component rises. Furthermore, due to the definite deployment of embedded software once it is released, it is unaffordable to deliver faulty software. A thorough testing is essential to minimize software bugs.

The design of embedded software is strongly dependent on the underlying hardware. Co-design of hardware and software is essential in a successful embedded system design. However, during the design time, the hardware might not always be available, so software testing is often impossible. Therefore testing is mostly postponed until after hardware development. Testing itself is typically limited to debugging or ad-hoc testing. Moreover, as it is the last phase in the process, it might be shortened when the deadline is nearing.

Integrating tests from the start of the development process is essential for a meticulous testing of the code. In fact, these tests can drive the development of software, hence Test-Driven Development.

In Section 2 we explain the TDD methodology and specific properties of embedded software development. Next, Section 3 illustrates how we migrated the principles of TDD to embedded software design and shows what pitfalls should be considered. In Section 4 we show some existing measurements available in literature on TDD in general software development. We projected these measurements to our experiments. Finally, Section 5 and 6 show related work and future work.

2 Problem statement

Subsection 2.1 gives a general overview of Test-Driven Development. Next, subsection 2.2 states the key properties of embedded software development.

2.1 Test-Driven Development

The software development process of Test-Driven Development (TDD) finds its origin in general software development. TDD's methodology originated in the late eighties, but since the general acceptance of cyclic development processes such as eXtreme Programming (XP), SCRUM and the Unified Process (UP), it has received more attention.

TDD is based upon the red-bar/green-bar mantra, which refers to the different colors of the software process steps. A TDD-cycle typically starts with

the selection of a specific requirement. This requirement is then translated into an executable test, as illustrated in Figure 1 in the block *Create Test*. Next, the minimal necessary skeleton code is implemented to make the test compile error-free, as illustrated in Figure 1 *Red Bar*. When the test fails, the developer knows that her test has detected unimplemented behavior. The test itself is correctly executing as designed. Next, the effective business code is implemented, which leads to a successful test, as illustrated in Figure 1 *Green Bar*. Finally, since during the development of the test and the implementation of the required behavior the focus was very narrow, the complete code for test and implementation is refactored numerous times. This refactoring, as illustrated in Figure 1 *Refactor*, does not add any extra behavior. It merely cleans the test code and business code from duplicates and badly implemented sections. This refactoring step is a safe step since all changes to the code are immediately verified by tests. Mind that the refactoring step should also revisit the test code, as improvements of test code are also needed.

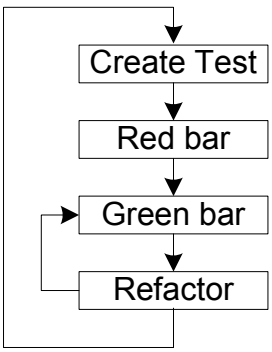

Figure 1: TDD Cycle

One of the main advantages of Test-Driven Development is that it shifts the focus of software creation to its interface, instead of its implementation. This is one of the first design principles mentioned by Gamma e.a. in [5], which improves the quality of the produced code. An analogous strategy was later noted by Kent Beck:

> By writing a test before implementing the item under test, attention is focused on the item's interface and observable behavior, Kent Beck [1]

A second advantage of test-driven development is that testing is considered early on in the development process. A developer will now start by thinking about these tests up front. As she interprets the specification of the required behavior, any misunderstandings about this specification will be detected early in the development process. This results in a cheaper way to correct

the specification or the expected behavior. As Boehm [2] notes in his graph, illustrated in Figure 2, if errors are detected later in the development process, they become more expensive to correct. In an embedded environment, it becomes almost impossible to correct an error once the system is in production because once the system is sold to customers, software patches are difficult to implement.

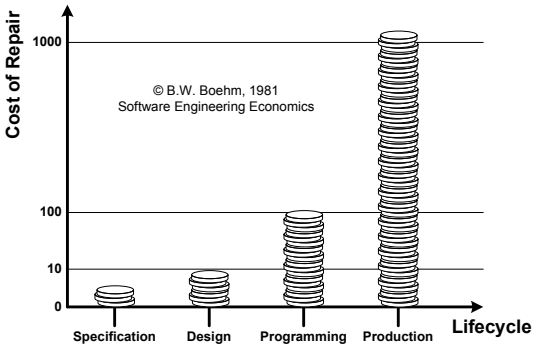

Figure 2: Cost of Repair in Software Engineering

Another advantage of Test-Driven development is that it focuses on automated testing. Tests should be written in such a way that they can be executed repeatedly and fully autonomously. Other advantages of TDD are: (1) Each test must execute in isolation of other tests, so one test does not influence the outcome of another. (2) The tests run under exactly the same circumstances every time. (3) The same test can be programmed to execute with many different values. This way the focus is not only on the current specific error, but all test cases are taken in consideration. (4) Automated tests can be run as many times as needed, whenever needed and even at regular intervals, e.g. when code is built. (5) The tests can also be called by a smoke-bot, which calls the tests every night for all the code that is currently checked-in in the repository.

2.2 Embedded Software Development

The term *embedded system* covers a wide range of electronic applications. These can be simple systems such as a controller for a microwave oven, or complex systems like a digital camera. However, all these systems have one thing in common, which is that they are designed with one specific application in mind. The embedded system looks like a small computer with dedicated hardware, but the main difference with a computer system is that its goal does not change during its lifetime. An embedded system for a digital camera will always remain the system for a digital camera, it will not

change into a system for a microwave oven. It should be noted that multiple embedded systems might be combined in one appliance. A cell phone could for instance integrate an mp3-player, as well as a digital camera. These underlying embedded systems can be fully integrated, or kept completely separate.

As mentioned by Vahid ea. [12], embedded systems have a number of key properties: (1) their restricted price, (2) their restricted size, (3) their required performance and (4) their restricted power consumption. More advanced properties are their reactiveness to events and their time-critical behavior. These properties make embedded system design a very specific co-design of hardware and software. The hardware developer must be aware of the software restrictions and vice versa.

It is crucial for embedded systems that they are tested very thoroughly, since the cost of repair grows exponentially once the system is taken in production, as stated in Section 2.1 Figure 2. However, the embedded system can only be tested once the complete development process is finished. Most embedded systems are developed using the waterfall process [11] for their software. First the user requirements are gathered. Next, these requirements are translated into functional specifications. Once all specifications are formally written down, the global technical design phase can start. After the global design comes the detailed technical design, as a basis for the next phase of programming. Finally, the system can be tested. If the hardware is still not available at this point, simulation tools and instruction set compilers are used to verify the behavior of the software component. Thorough testing can only be done when the hardware is fully configured. In the current strategy for developing embedded systems, the testing phase is generally done manually. This ad-hoc testing is mostly heuristic and only focuses on one specific scenario. At this point debugging facilities are very handy to look at the inside functioning of the software component that is tested. As noted, the testing is done late in the development cycle, with all due disadvantages.

When we want to start testing as early as possible, a number of problems arise. One problem being the hardware unavailability, and another being the difficulty to automatically test embedded systems. The following Section 3 illustrates techniques to tackle these problems.

3 Migration of Test-Driven Development Principles to Embedded Software Development

During the experiments in preparation of this paper, two strategies were used for creating and building an embedded system using the Test-Driven

Development technique. In Section 3.1 we describe how we created the system performing the tests in the host environment. Next, in Section 3.2 we describe how we created the system performing the tests on the target environment.

3.1 Testing in Host Environment

Figure 3 illustrates the setup for development. The software for the embedded system, noted as *program* in the illustration, was developed using a TDD strategy based on virtual drivers. An extra indirection was added to the code by using an interface, as this allowed us to replace the virtual drivers by real drivers when hardware and drivers became available. Once the software was stable, all virtual drivers were replaced by the real drivers and tests could be run on the embedded system.

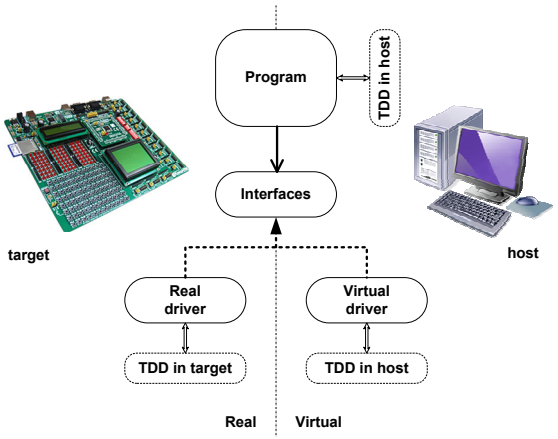

Figure 3: Testing in Host Environment

This configuration has a number of advantages. First of all we could use existing tools for test-driven development. In this case an existing C++ unit framework was used on the host environment. Integration of this framework in development tools gave us the second advantage of fast feedback. Since all code was created and tested on the same host environment, the typical TDD-mantra red/green/refactor was applied frequently.

The configuration also has some drawbacks. Firstly, there is no guarantee that the virtual drivers effectively mimic the real drivers. Secondly, while the development program runs on an equivalent environment, it is not certain that the code will work on the effective hardware board. Hence, cross compiling issues may arise.

3.2 Testing in Target Environment

A second setup for development is illustrated in Figure 4. In this configuration, the software component was developed in the host environment. The testing cycle now includes the migration of the component to the target environment. In this target environment the real drivers are used and an extra testing wrapper is installed. This wrapper allows the software component on the hardware board to be tested from the host environment.

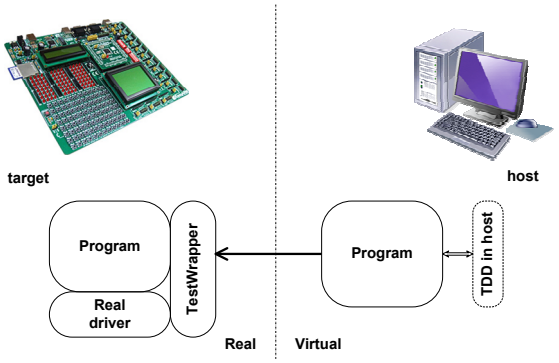

Figure 4: Testing in Target Environment

Advantages of this configuration are that the real compiler is used, so no cross compiling issues will arise. Secondly, all tests are now performed on the effective hardware board, so the real environment is now in use for the tests.

The most important drawback of this configuration is that the test cycles are now noticeably longer. Each time the tests must be run, the software component must be deployed to the board first. As opposite to the previous configuration of Section 3.1, this configuration is very difficult to fully automate. Steps to automate the process are: migrating the component, starting the board, starting the test bench on the host environment and inspecting the results. Integrated development environments for embedded systems with this strategy of TDD in mind do not exist. Finally, the test wrapper requires special care, so that it functions correctly and that its memory footprint does not exceed the available memory.

3.3 Specifications

As mentioned in Section 2.2, embedded systems are generally designed following a waterfall development process. The change that we propose to this process is that the specifications, formally written down before we start the global technical design, can be used as a basis for the creation of tests. This

way, the developer is pushed early on in the development process to pay attention to writing tests. Once the system is ready for production, these tests created on basis of the specifications can act as acceptance tests.

3.4 Beware of Software Testing Pitfalls

Developing using the TDD strategy should be done with caution. A pitfall generally known as *the broken window syndrome* states that one must be careful with broken tests. The basic idea of broken window comes from windows in an abandoned building: once one test starts to fail, shortly after the number of failing tests will grow gradually, ultimately leading to all tests being broken. The tests can break for different reasons: the expected behavior is changed, or the implementation of the business code is changed. In the former case the developer adjusts the test code to correctly test the updated expected behavior. In the latter case the implemented business code does not support the original expected behavior. To solve this problem, the business code is adjusted so the test succeeds.

Martin Fowler [4] states *test cancer* as another pitfall. He states that it is a bad habit to leave out a failing test from the test scenario. One could think that by leaving out this test, the scenario now completely succeeds. But when the failing test is not corrected, more and more tests will fail and be left out from the scenario. This way the failing tests spread like cancer through the developed code.

A final pitfall is the fact that while developing more and more tests to check the expected behavior, one could think that the created system is free of bugs. But testing does not prove that a system is bug free, or as Edsger Dijkstra stated in ACM:

> Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. [3]

4 Measurements

In this section we illustrate different measurements of using test-driven development to create software. First measurements of Müller [9] and Microsoft Research [10] are explained, later we show our own experiences found during the experiments of Section 3.1 and 3.2.

As illustrated in Figure 5 the development phase of test-driven development takes longer than that of conventional development. A reason for this is the extra effort it takes to create tests during development. However, a quality assurance phase is necessary when creating software using conventional

development processes. During quality assurance, bugs and misconceptions are fixed in the software. This phase is generally known as the beta test phase. This quality assurance phase is spread out during TDD, hence the longer development time. The investment in testing early on in TDD results in a non-negligible life cycle benefit

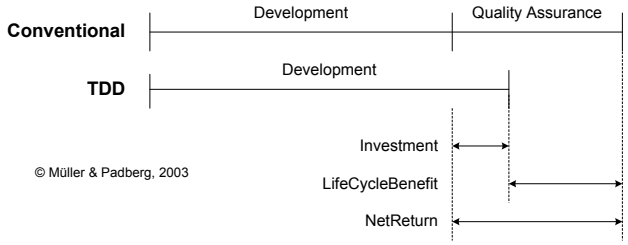

Figure 5: Return on Investment of Test-Driven Development [9]

Table 1 illustrates the results of experiments conducted by Microsoft Research [10]. The experiments are based on two comparable software development teams. One team uses its current development strategy to create a specific component, the other team was first trained in TDD and then used this TDD strategy to develop the same component. Table 1 compares the pre-release defect density in the end result of both teams. The pre-release defect density measures how much effort is necessary to make the software component function correctly after its creation. The third row shows that in all four experiments the defect density of the TDD team was lower. This reduction is typical for TDD and comes along with a greater software quality of test-driven code. The reason for the reduction is that when tests are used early in the development, some errors appear more quickly and are fixed earlier than in traditional software development.

Table 1: Pre Release Defect Density

Metric description	**IBM Drivers**	**Microsoft Windows**	**Microsoft MSN**	**Microsoft VS**
Defect density of comparable team in organization but not using TDD	W	X	Y	Z
Defect density of team using TDD	0.61W	0.38X	0.24Y	0.09Z
Increase in time taken to code the feature because of TDD (%) [Management Estimates]	15-20%	25-35%	15%	20-25%

Finally, our own experiments on embedded systems confirmed the previous measurements. On the one hand, they showed that we needed extra time to

develop the virtual drivers. On the other hand, these virtual drivers allowed us to create the software component before all hardware was available. This allowed us to do in place detection of bugs and correction of behavioral faults even before the real drivers were available.

5 Related Work

The two approaches as described in Section 3, are reflected in the work of respectively Grenning [6] and the people of Atomic Object [7, 8]. The common ground from which both start the implementation of TDD on embedded software. Both recognize the automation of unit tests as the most important feature to reach the objective. Without an automated test suite, tests take too long to execute or require too much effort and quickly get abandoned by the developer. However, the main point of difference, is the implementation of the automated test suite.

Grenning [6] proposes the embedded TDD cycle, which resembles the previously described methods of developing the software on the host and then periodically checking the code and tests on the target. Another resemblance is the strong belief that Object Oriented principles can also be applied to embedded systems, despite memory issues.

Karlesky ea. [7, 8] do not want to add Object Oriented overhead in the embedded software. Therefore they created the Model Conductor Hardware pattern, which is a naming convention for C in embedded systems isolating the hardware from the business logic. To test the different components of the pattern, mock versions were created. Also, they implemented the pattern directly into the target. To achieve a reasonable turnaround of the test results, they automated the entire process, of downloading, calling, executing and returning the test results. They also found that the manual creation of mocks is too tedious and prone to errors. To solve this problem they made an automatic mock generation script and integrated it into the development environment. This in turn, had the disadvantage that many mocks were created that ended up unused, which put extra unnecessary strain on the memory of the system.

6 Future Work

The described experiments already contained testing of the individual components in the larger embedded system. Further steps are necessary to test the complete integrated system. The problem that arises here is the non-deterministic behavior of the complete system. To write tests in such a non-

deterministic setup, the focus should be shifted to the deterministic parts properties of the system. For instance, a sensor may return a valid result for 97% of the time. Such a sensor makes the system non-deterministic since we might not know whether the correct value should be tested, or whether the code should be tested that ignores the measurement, since its value is out of range. Both paths should be tested, in such a way that the correct path is tested 97% of the time and the ignoring path is executed 3%.

Also, real-time code still poses a problem for the integration of automated testing. Issues like concurrency and timing constraints have not been addressed to date, but this will be necessary if TDD is to be successful in the whole range of embedded systems.

Definite measurements comparing the development of two similar projects, one using TDD and the other not, should show (1) a decrease in development time and (2) earlier detection of bugs.

7 Conclusion

This paper shows that integrating Test-Driven Development in embedded software design is feasible. However, one must try to automate at least a major part of the testing cycle. This asks for serious commitment of the development team, but, as shown in Section 4, the benefits of TDD outweigh its cost. We illustrated that running tests in both the host and target environment poses problems, amongst others cross-compiling issues. However, this approach covers a broader range of testing scenarios and facilitates the frequent running of the test suite. Finally, we confirmed that the existing measurements of TDD in general software development also apply to embedded software development.

References

[1] K. Beck. *Test Driven Development: By Example.* Addison-Wesley Longman Publishing, Boston, MA, USA, 2002.

[2] B.W. Boehm. *Software Engineering Economics (Prentice-Hall Advances in Computing Science& Technology Series).* Prentice Hall PTR, October 1981.

[3] E. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[4] M. Fowler. Test cancer. *http://martinfowler.com/bliki/TestCancer.html* (Accessed 11/09), December 2007.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing, USA, 1995.

[6] J. Grenning. Test driven development for embedded software. In *Embedded Systems Conference (ESC-241)*, 2007.

[7] M. Karlesky, W. Bereza, and C. Erickson. Effective test driven development for embedded software. In *2006 IEEE International Conference on Electro Information Technology*, pages 382–387+, 2006.

[8] M. Karlesky, G. Williams, W. Bereza, and M. Fletcher. Mocking the embedded world: Test-driven development, continuous integration,and design patterns. In *Embedded Systems Conference (ESC-413)*, 2007.

[9] M. Müller and F. Padberg. About the return on investment of test-driven development. In *International Workshop on Economics-Driven Software Engineering ResearchEDSER-4*, 2003.

[10] N. Nagappan, M. Maximilien, T. Bhat, and L. Williams. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering*, 13:289–302, 2008.

[11] I. Sommerville. *Software Engineering: (Update) (8th Edition)*. Addison Wesley, 8 edition, June 2006.

[12] F. Vahid and T. Givargis. *Embedded System Design: A Unified Harware/Software Introduction*. John Wiley & Sons, 2002.